Q1 Directions/assurance

0 Points

1. Please read the entire exam before starting to write. This should help you avoid getting bogged down on one problem.
2. We believe this is approximately two "exam hours" of content. However, you will have four hours to work on the exam, starting from when you begin it. The extra time is intended to correct for potential logistical issues and also to reduce stress during a time when we all likely have more than we need.
3. The exam will be open-book/open-notes in the following sense:
   A. You may use any edition of either textbook.
   B. You may refer to materials we provided you with, including the Intel PDFs on the course web site, the lecture slides we provided you with, the project handouts (the kernel specification and the thread-library handout may be particularly useful), and the lecture videos from this semester---in other words, materials on this semester's course web site.
   C. You may use notes you have taken.
   D. You may refer to your P0, P1, and P2 submissions.
4. But you are **not** authorized to get help from other people, or to use online resources that are not part of this semester's course web site.
5. You are **not** authorized to use compilers, assemblers, linkers, loaders, simulation engines, virtualization systems, proof checkers, etc.
6. The exam will not be proctored, i.e., you will be operating on the honor system in terms of resources we have said you can consult.
7. If you have a question while taking the exam, please check the published Zoom schedule. You may also send mail to the staff mailing list.
8. The weight of each question is indicated on the exam. Weights of question **parts** are estimates which may be revised during the grading process and are for your guidance only.
9. Please be concise in your answers. You will receive partial credit for partially correct answers, but truly extraneous remarks may count

against your grade.

I certify that my exam submission is my own work, in compliance with the rules stated above.

Name:

Date:

Q2 Error kinds

4 Points

According to the 15-410 orthodoxy, there are three kinds of error. Please present, based on your P2 thread library, one example each of **two** of the three kinds of error. For each example, briefly describe what the code was trying to do, how the failure qualifies as that particular kind of error, and the action that your P2 implementation should have taken. You do not need to show actual code (we are interested in your description/analysis).

Q2.1 Kind 1

2 Points

Description:

Q2.2 Kind 2

2 Points

Description:

Q3 Register dump

4 Points

Below is a register dump produced by the "Pathos" P2 reference kernel when it decided to kill a user-space thread. Your job is to carefully consider the register dump and:

1. Determine which "wrong register value(s)" caused the thread to run an instruction which resulted in a fatal exception. You should say why/how the wrong value led to an exception, i.e., merely claiming a register has a "wrong" value will not receive full credit.
2. Briefly state the most plausible way you think the register(s) could have taken on the value(s) (i.e., try to describe a bug which could have this effect).
3. Then write a **small** piece of code which would plausibly cause the thread to die in the fashion indicated by the register dump. **This code does not need to implement exactly the set of steps that you identified as "most plausible" above, or result in precisely the same register values; you should aim to achieve "basically the same effect."** Most answers will probably be in assembly language, but C is acceptable as well. Your code should assume execution begins in `main()`, which has been passed the typical two parameters in the typical fashion.

Please be sure that your description of the fatality and the code, taken together, clearly support your diagnosis.

```
Registers:
eax: 0x00000000, ebx: 0x00000000, ecx: 0xffffeec4,
edx: 0xffffefc4, edi: 0x00000000, esi: 0x00000000,
```

```
ebp: 0x01000066, esp: 0x01000066, eip: 0x01000029,
 ss:     0x002b,  cs:     0x0023,  ds:     0x002b,
 es:     0x002b,  fs:     0x002b,  gs:     0x002b,
eflags: 0x00000282
```

Wrong register value(s) and exception/fault reason:

How the register(s) got the wrong value(s):

Code: if possible, please upload a text file containing your code, ideally with spaces rather than tabs. But we will also accept PDF or PNG -- if possible, please provide us with black text on a white background, and please avoid white text on a black background or stranger things (purple text on a black background, etc.).

No files uploaded

Q4 Synch or Swim

15 Points

After a particularly harrowing battle with a giant squid, Captain Nemo decides to jump the shark and upgrade his 19th-century submarine, the **Naughtilus**, with modern technology. Unfortunately, Nemo has been underwater for a long time (with a 1st-edition OSC textbook), and so he is relying on your expertise to fix his faulty code. Specifically, Nemo needs you to address his concerns regarding the **Naughtilus**'s airlock system, which is detailed below.

The airlock has two doors. The "inner" door leads into the submarine and the "outer" door leads out underwater. The doors are activated by two void functions, `open_inner_door()` and `open_outer_door()`. These functions immediately and unconditionally open the indicated doors,

without any safety checks (and without being impeded by the physics of water pressure). Note that if the submarine is underwater then opening both doors at the same time will instantly flood the submarine with seawater, killing the entire crew.

The airlock system involves two fluids, water and air, which will be represented thus:

```
#define AIR 1
#define WATER 0
```

A powerful pump can force air into the airlock while forcing water out, or can suck air out of the airlock while allowing water to enter. Pumping can take substantial time, so a callback approach is used. When pumping is complete, an interrupt handler will invoke a callback function with one parameter, indicating whether the airlock now contains air or water. Thus the signature is:

```
typedef void (*callback_t)(int result_fluid);

void start_pump(callback_t callback, int desired_fluid);
```

If you wish, you may assume that the pumping system can queue an infinite number of requests, which it will act on in some order, which you may assume to be FIFO if you wish.

Because Captain Nemo doesn't trust his crew with a low-level API, the only functions exposed to crew members are `enter_submarine()` and `exit_submarine()`.

Consider the following code.

```
int atmosphere = AIR; // is the airlock chamber filled with air, o
cond_t airlock_vent; // used to wait on the airlock to compress or
mutex_t vent_control_lock; // arbitrates access to the ventilation
mutex_t inner_door_lock; // arbitrate access to the inner door
mutex_t outer_door_lock; // arbitrate access to the outer door
```

```
 /** Handle compress/decompress completion interrupts
  *  set atmosphere and signal the waiter
  */
1.   void compress_interrupt_handler(int result_fluid) {
2.       mutex_lock(&vent_control_lock);
3.       atmosphere = result_fluid;
4.       cond_signal(&airlock_vent);
5.       mutex_unlock(&vent_control_lock);
6.    }

 /** Compress/decompress the airlock
  *
  *  Issue a request to (de)compress the airlock and
  *  cond_wait() for fulfillment
  */
7.  void compress(int desired_fluid) {
8.      mutex_lock(&vent_control_lock);
9.      start_pump(&compress_interrupt_handler, desired_fluid);
10.     while(atmosphere != desired_fluid){
11.          cond_wait(&airlock_vent, &vent_control_lock);
12.     }
13.     mutex_unlock(&vent_control_lock);
14.  }

  /** Protocol for entering the submarine.
   *
   *  First, flood the airlock chamber and open the outer door.
   *
   *  Then close the outer door, compress the chamber with air,
   *  and open the inner door.
   */
15.  void enter_submarine(void) {
16.      // make sure we don't flood the inside of the submarine
17.      mutex_lock(&inner_door_lock);
18.      compress(WATER);
19.      mutex_lock(&outer_door_lock);
20.      open_outer_door();
21.      sleep(2); // time for crew to swim into the airlock
22.      close_outer_door();
23.      mutex_unlock(&outer_door_lock);
24.      compress(AIR);
25.      open_inner_door();
26.      sleep(4); // time for crew to enter the submarine
27.      mutex_unlock(&inner_door_lock);
28.   }

   /** Protocol for exiting the submarine.
    *  First, compress the airlock chamber and open the inner door
    *  Then close the inner door, decompress the chamber,
```

```
    *  and open the outer door.
    */
29.  void exit_submarine(void) {
30.      // make sure we don't flood the inside of the submarine
31.      mutex_lock(&outer_door_lock);
32.      compress(AIR);
33.      mutex_lock(&inner_door_lock);
34.      open_inner_door();
35.      sleep(1); // time for crew to step into airlock
36.      close_inner_door();
37.      mutex_unlock(&inner_door_lock);
38.      compress(WATER);
39.      open_outer_door();
40.      sleep(4); // time for crew to swim out of airlock
41.      mutex_unlock(&outer_door_lock);
42.  }
```

Assume all global synchronization variables begin initialized and unlocked. If you wish, you may assume that this code is running on a uniprocessor machine.

Q4.1 Multi-thread problem

12 Points

Show, via a tabular execution trace of the form used in this class, how a deadlock involving **two or more** threads can occur in this system.

First **briefly but clearly** summarize the deadlock.

Now show a **clear and compelling** execution trace. We will accept a picture (PDF, PNG), or a text file containing one line per event (each event should consist of a thread i.d. and an action, e.g., `T0: x=3`). **Please avoid color combinations that are difficult to read**, such as purple text on a black background.

No files uploaded

Q4.2 Second problem
3 Points

Show, via a tabular execution trace of the form used in this class, how a deadlock involving **just one** thread(!?!) can occur in this system.

First **briefly but clearly** summarize the deadlock.

Now show a **clear and compelling** execution trace. We will accept a picture (PDF, PNG), or a text file containing one line per event (each event should consist of a thread i.d. and an action, e.g., `T0: x=3`). **Please avoid color combinations that are difficult to read**, such as purple text on a black background.

No files uploaded

Q5 Semaphores
20 Points

Columbus Unix was an early-1980's version of Unix that Bell Labs provided to Bell System product groups, some of whom used it as the foundation for important telephone equipment. Columbus Unix provided kernel-implemented semaphores to user-space programs. According to legend, the in-kernel implementation of semaphores suffered from a grievous design flaw: if multiple processes blocked on one semaphore, when the semaphore was signaled, processes were awakened in **LIFO** order (not FIFO order)! The legend states that this grievous flaw was not noticed in production because the specific high-reliability telephony applications using the kernel-provided semaphores never had more than one process at a time blocked on any given semaphore.

Now that you have learned this, you probably can't forget it, and you are probably wondering how many semaphore implementations might make this mistake. Unfortunately, not all semaphore implementations are open-source, and some are locked away inside a kernel. But you are an adept

thread programmer, so you decide to write some test code that will probe the operation of a semaphore implementation. The mission of your test code is to, within a matter of seconds, print either "LIFO" or "FIFO" (but not both) and then complete.

Of course, it is extremely difficult for a simple test program to **completely** determine whether a black-box semaphore implementation is LIFO or FIFO, but the goal of your test code is to catch with high probability (well over 95%) implementations that are consistently broken. It is acceptable if your program prints either "LIFO" or "FIFO" multiple times, as long as all of the printed strings agree. It is also permissible for your program to print "FAIL", or to invoke `assert()` or `panic()`, if it detects an impossible outcome, though you are **not** required to check for inconsistencies (see below). What your test program does after printing its verdict doesn't matter: your test program is allowed to hang, crash, etc., because the printed answer will be reviewed by a live human being.

Your semaphore-test code can use mutexes, condition variables, and/or reader/writer locks, and you can assume that they are correctly implemented. If you wish, you may assume that condition variables awaken threads in strictly-FIFO order. Note that, as an interface-compliant test program, *you cannot inspect or modify the internals of any thread-library data objects*--because your test code must run against any semaphore implementation, you are strictly a *client* of the abstraction.

However, because this is test code, you **may** employ normally-distasteful code constructs such as yield loops, or even spin-waiting if you must; that said, solutions with smaller quantities of distasteful code are likely to receive higher scores.

Other notes:

1. You may refer to the Pebbles kernel specification and the Project 2 library specification.
2. Assume that `sem_init()`, `sem_signal()`, etc., operate on user-space semaphore objects regardless of whether the implementation of the those operations is user code or kernel code.

3. You may not use assembly code, inline or otherwise.
4. **For the purposes of the exam, you may assume that library routines and system calls don't "fail"** (unless you indicate in your comments that you have arranged, and are expecting, a particular failure).
5. You may **not** rely on any data-structure libraries such as splay trees, red-black trees, queues, stacks, or skip lists, lock-free or otherwise, **unless you implement that code as part of your submission**.
6. Your program should include a `main()` function, which presumably will initialize and operate on at least one `sem_t` object.

Good luck!

Please upload a single text file containing your semaphore-test code. Ideally the text file will use spaces rather than tabs. But we will also accept PDF or PNG -- if possible, please provide us with black text on a white background, and please avoid white text on a black background or stranger things (purple text on a black background, etc.).

```
sem_t s;

int main(void)
{
    print("FAIL");
    return(99);
}
```

Code file:

No files uploaded